FChain: Toward Black-box Online Fault Localization for Cloud Systems

Hiep Nguyen[§], Zhiming Shen[§], Yongmin Tan[‡], Xiaohui Gu[§]

[§]*North Carolina State University, Raleigh, NC*

[§]{hcnguye3, zshen5}@ncsu.edu, gu@csc.ncsu.edu

[‡]*MathWorks Inc.*

[‡]yongmin.tan@mathworks.com

***Abstract*—Distributed applications running inside cloud systems are prone to performance anomalies due to various reasons such as resource contentions, software bugs, and hardware failures. One big challenge for diagnosing an abnormal distributed application is to pinpoint the faulty components. In this paper, we present a black-box online fault localization system called *FChain* that can pinpoint faulty components *immediately* after a performance anomaly is detected. FChain first discovers the onset time of abnormal behaviors at different components by distinguishing the *abnormal* change point from many change points caused by normal workload fluctuations. Faulty components are then pinpointed based on the *abnormal change propagation* patterns and *inter-component dependency* relationships. FChain performs runtime validation to further filter out false alarms. We have implemented FChain on top of the Xen platform and tested it using several benchmark applications (RUBiS, Hadoop, and IBM System S). Our experimental results show that FChain can quickly pinpoint the faulty components with high accuracy within a few seconds. FChain can achieve up to 90% higher precision and 20% higher recall than existing schemes. FChain is non-intrusive and light-weight, which imposes less than 1% overhead to the cloud system.**

I. Introduction

Infrastructure as a Service (IaaS) clouds [1], [2] allow multiple tenants to share a common physical computing infrastructure in a cost-effective way. However, applications running inside the IaaS cloud are prone to performance anomalies such as service level objective (SLO) violations due to various reasons such as resource contentions, software bugs, or hardware failures. It is particularly challenging to localize the faulty components in a complex distributed application that consists of many inter-dependent components.

Previous work on distributed system debugging can be categorized as white-box, grey-box, or black-box techniques. White-box and grey-box techniques (e.g., [3]–[5]) require modifications or instrumentations to applications or underlying middleware platforms. Those intrusive techniques often impose significant runtime overhead and are difficult to deploy, which make them impractical for performing *online* fault localization in the production IaaS clouds. Existing black-box techniques [6]–[8] mostly focus on anomaly detection or system metric attribution. In contrast, our goal is to pinpoint faulty components in a distributed application. Other schemes [9]–[11] only work for certain types of faults or applications. We will discuss related work in detail in Section IV.

In this paper, we present FChain, a black-box online fault localization system for diagnosing performance anomalies in IaaS clouds. FChain can pinpoint faulty components *immediately* after a performance anomaly is detected. FChain does not perform any intrusive tracing in the cloud system and only relies on low-level system metrics (e.g., CPU, memory, network statistics) that can be easily obtained from the guest OS or hypervisor. FChain does not assume any prior application knowledge, which makes it practical for IaaS clouds. Moreover, FChain does not require any training data for anomalies, which can diagnose both previously seen and unseen performance anomalies. Figure 1 shows the overall architecture of the FChain system.

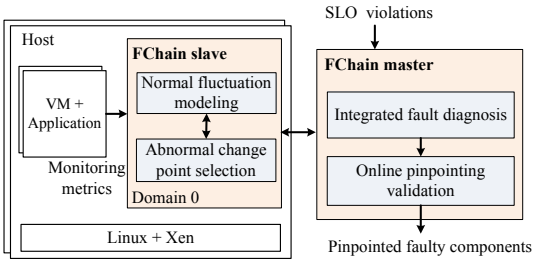

Fig. 1. The overall architecture of the FChain system. The FChain slave continuously collects system-level metrics (e.g., CPU, memory) on different application VMs running inside the cloud and learns normal fluctuation patterns. When a SLO violation is detected, the FChain master triggers all related FChain slaves to look for abnormal change points. The FChain master then pinpoints faulty components based on the change point timing information and inter-component dependencies. FChain also performs online validation to remove possible false alarms.

When a performance anomaly is detected, FChain first discovers abnormal changes at all system metrics of different components. Next, FChain extracts abnormal change propagation paths for the diagnosed performance anomaly and localizes the faulty components based on the abnormal change propagation patterns. Our design is based on two key observations: 1) performance anomalies often manifest as abnormal system metric fluctuations that are distinctive from the normal fluctuation patterns; and 2) the abnormal system metric changes often start from the faulty components and then propagate to other non-faulty components via inter-component interactions.

To achieve robust fault localization, FChain needs to address a challenging problem: how to distinguish the abnormal change point that marks the onset of the fault manifestation from many other change points that are caused by normal workload fluctuations. It is insufficient to use a fixed filtering threshold since some applications are inherently more dynamic than others. To address this problem, FChain captures the normal fluctuation patterns using online system metric value prediction models [12] and uses a *predictability* metric to iden-

tify abnormal fluctuations. As we will show later in Section III, this abnormal change detection scheme can achieve higher accuracy than traditional anomaly detection schemes [10].

Due to inter-component interactions, abnormal changes in the faulty component(s) often propagate to other normal components. If we examine each component in an isolated way, we might produce many false alarms by mistakenly pinpointing normal components as faulty ones. Our previous work [13] has shown that it is feasible to pinpoint faulty components by sorting all affected components based on their fault manifestation time and identifying the component with the earliest manifestation time as the faulty one. However, we observe that it is insufficient to only rely on the chronological order to perform fault localization since we may derive spurious abnormal change propagations between two independent components. To address the problem, FChain integrates the dependency relationships with the fault propagation model to achieve more accurate pinpointing than existing schemes.

This paper makes the following contributions:

- We present FChain, a practical *online* fault localization system for large-scale IaaS clouds. FChain does not require any intrusive application monitoring, which can diagnose both previously seen and unseen anomalies.
- We describe a predictability-based abnormal change point selection scheme that can distinguish abnormal change points that are related to the fault manifestation from those normal change points that are caused by normal workload fluctuations.
- We introduce a new integrated fault localization scheme considering both fault propagation patterns and inter-component dependencies to achieve higher pinpointing accuracy.

We have implemented FChain on top of the Xen platform and tested it on NCSU's Virtual Computing Lab (VCL) [2], a production cloud computing system that operates in a similar way as Amazon EC2 [1]. We conducted extensive experiments using a set of common faults and different types of applications (IBM System S data stream processing system [14], Hadoop [15], and RUBiS online auction benchmark [16]). Our experimental results show that: 1) FChain can achieve up to 90% higher precision and 20% higher recall than existing black-box fault localization schemes; 2) FChain can complete the fault localization within a few seconds and works for different types of applications; and 3) FChain is light-weight, imposing less than 1% CPU overhead during system runtime execution.

The rest of the paper is organized as follows. Section II describes the design of the FChain system. Section III presents our experimental evaluation. Section IV compares our work with related work. Finally, the paper concludes in Section V.

II. System Design

In this section, we first present an overview of the FChain system. We then describe the abnormal change point identification algorithm and the integrated faulty component pinpointing scheme.

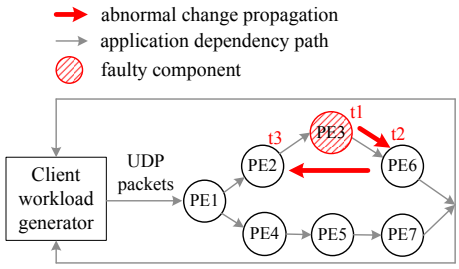

Fig. 2. Abnormal change propagation in an IBM System S application. The fault propagates along the path: $PE_3 \rightarrow PE_6 \rightarrow PE_2$. The propagation $PE_6 \rightarrow PE_2$ is caused by the back-pressure effect.

A. System Overview

FChain is decentralized consisting of a set of slave modules (i.e., normal fluctuation modeling, abnormal change point selection) and master modules (i.e., integrated fault diagnosis, online pinpointing validation), shown by Figure 1. The slave modules run inside the domain 0 of different cloud nodes while the master modules run on dedicated servers. FChain treats each guest virtual machine (VM) as one component.

The normal fluctuation modeling module continuously monitors the system metrics for each VM to capture its normal fluctuation pattern. We employ a light-weight online learning model [12] to continuously learn the evolving pattern of each system metric value. If the change is caused by normal workload fluctuations, the prediction model must have seen and learned the change before. Thus, the prediction errors on those normal change points will be small. In contrast, the fluctuations caused by faults are not captured by the online learning model, which will probably incur high prediction errors.

When a performance anomaly (e.g., SLO violations) is detected[1], the FChain master is invoked to pinpoint the faulty components in the failing distributed application. The FChain master first contacts the slaves on all related distributed hosts to identify whether a component exhibits abnormal changes and when the abnormal change begins. If the performance anomaly occurs at time t_v, the FChain slaves check a window $([t_v - W, t_v])$ (e.g., $W = 100$) of recent metric values before t_v. The abnormal change point selection module uses the normal fluctuation model to filter out those change points that are caused by normal workload fluctuations.

The integrated fault diagnosis module comprehensively examines the abnormal change point information from all components and the inter-component dependency information to pinpoint the culprit component(s). FChain first derives the abnormal change propagation pattern in the examined distributed application by sorting the timestamps of the abnormal

[1]Note that FChain focuses on fault localization rather than performance anomaly detection that has been addressed by previous work(e.g., [17], [18]).

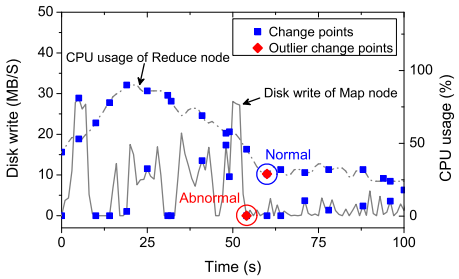

Fig. 3. Abnormal change point selection for the DiskWrite metric of a faulty map node and the CPU usage metric of a normal reduce node in a Hadoop sorting application.

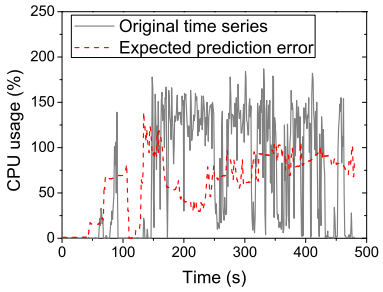

Fig. 4. Expected prediction error for the CPU usage metric on a dual-core host.

change points of different components.[2] For example, Figure 2 shows the abnormal change propagation path in a sample IBM System S application consisting of seven distributed components called processing elements (PEs). The application consists of one faulty component PE_3 with a memory leak bug. The memory usage metric of PE_3 shows the abnormal change at time t_1. The anomaly first propagates to the component PE_6 at time t_2 and then to the component PE_2 at time t_3. Since $t_1 < t_2 < t_3$, FChain pinpoints PE_3 as the suspicious root cause component. Note that our scheme does not assume any knowledge about the application including its topology. FChain leverages the black-box dependency discovery tool [11] to filter out spurious abnormal change propagations between independent components.

Finally, FChain performs online pinpointing validation using the dynamic resource scaling technique in a similar way as [20]. Since FChain monitors various system metrics, it can not only pinpoint faulty components but also identify which system metrics are related to the fault. We can then adjust those metrics on the faulty components to validate the accuracy of the pinpointing results by observing the resource adjustment impact to the application's SLO violation status.

B. Abnormal Change Point Selection

To localize the faulty components, FChain first examines i) *which components* exhibit abnormal changes in different system-level metrics; and ii) *when* those abnormal changes start. As mentioned before, system-level metrics are inherently fluctuating under dynamic workloads. If we apply standard change point detection algorithms, we often discover many change points. For example, Figure 3 shows the set of change points discovered on the "Disk Write" metric of a map task node and the "CPU usage" metric of a reduce task node in a Hadoop application, using the common change point detection algorithm "CUSUM + Bootstrap" [21]. We can see many change points are just random peak and bottom values, which are not actually related to the fault. We can use smoothing and change magnitude outlier detection to filter some normal change points [13]. However, it is insufficient to only rely on smoothing and outlier detection to select abnormal change points since some metrics (e.g., the "Disk Write" metric in Figure 3) have large variations during normal execution.

FChain uses a predictability metric to achieve robust abnormal change point selection. We employ an online learning model [12] to continuously learn the change pattern of each system metric. The online learning model can capture the transition probability between different metric values using a discrete time Markov chain model. Intuitively, the value transition patterns at a normal change point should be able to be captured by the online learning model and thus easier to predict. We calculate a prediction error for each outlier change point by comparing the predicted value with its true value. If the prediction error is high, we consider this outlier change point as an abnormal change point. However, it is a non-trivial problem to pick a proper prediction error threshold for filtering normal change points since some metrics (e.g., bursty ones) are inherently harder to predict than others and vary from application to application. Thus, it will be imprecise to apply a fixed filtering threshold.

To address this problem, FChain dynamically computes a proper prediction error threshold for each change point based on the burstiness of the time series surrounding the change point. The intuition behind our scheme is that a bursty metric is expected to have a higher prediction error than a non-bursty metric. Thus, we want to use a higher prediction error threshold when the metric values are bursty. Specifically, we extract a small window of time series data surrounding the change point x_t: $X = x_{t-Q}, ..., x_{t+Q}$ (e.g., $Q = 20$ seconds) and apply the fast Fourier transform (FFT) algorithm on X to determine the coefficients that represent the amplitude of each frequency component. We consider the top k (e.g., 90%) frequencies in the frequency spectrum as high frequencies. We apply inverse FFT over the high frequency components to synthesize the burst signal. We then use the burst signal magnitude (e.g., 90th percentile of the burst value) as the expected prediction error for the change point x_t. If the real prediction error exceeds the expected prediction error, the change point is selected as one abnormal change point. For example, Figure 4 illustrates the expected prediction errors for a system metric time series. We can see that the expected prediction error is higher when the original time series are

[2]Since FChain relies on timing information to infer abnormal change propagations, we synchronize the clocks of all hosts using the network time protocol (NTP), which has an error of less than 0.1 ms in LANs and less than 5 ms on the Internet [19]. In our experiments, we observe that all of the anomaly propagation delays between two dependent components are at least several seconds. Therefore, our system can tolerate small time skews (i.e., tens of milliseconds).

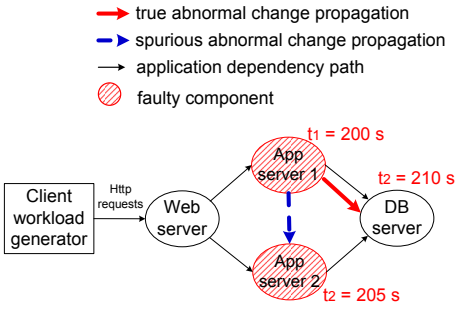

Fig. 5. Faulty component pinpointing for the RUBiS online auction benchmark application.

bursty, and is lower when the time series become stable. In Figure 3, our scheme correctly filters out the outlier change point on the normal reduce node and only selects the abnormal change point on the faulty map node.

After identifying a component exhibiting any abnormal change, we need to know when the abnormal change starts. During our experiments, we found that the selected abnormal change point sometimes resides in the middle of the fault manifestation process instead of at the beginning, depending on the evolving pattern of the fault manifestation (e.g., gradual change or bursty change). Thus, FChain performs tangent-based rollback to identify the precise start time of the abnormal change. Specifically, starting from the abnormal change point, we compare the tangent of the current change point with that of its preceding change point. If their values are close (e.g., < 0.1), we roll back to the preceding change point. We repeat the roll-back process until the tangents of two adjacent change points are distinct. If a component has multiple metrics exhibiting abnormal changes, we pick the earliest abnormal change start time as the component's abnormal change start time.

C. Integrated Faulty Component Pinpointing

Our faulty component pinpointing algorithm consists of three steps: 1) deriving the abnormal change propagation paths in the examined distributed application by sorting the start time of abnormal changes at different components; 2) pinpointing faulty components based on the selected abnormal change propagation paths; and 3) refining pinpointing results by filtering out spurious abnormal change propagation paths using inter-component dependency information. For example, Figure 5 shows the pinpointing process for the RUBiS online auction benchmark application [16].

If the abnormal change onset time of C_1 is earlier than that of C_2, we say that the abnormal change propagates from the component C_1 to the component C_2. For example, in Figure 5, the application server 1 starts to exhibit abnormal change at time $t_1 = 200$ seconds and the database server starts to show abnormal change at time $t_3 = 210$. Thus, we can infer that the abnormal change starts at the application server 1 and propagates to the database server.

To pinpoint faulty components, FChain first sorts all the components into a chain based on their fault manifestation time. We first pinpoint the source component in this chain as the faulty component since it has the earliest fault manifestation. We continue to examine the other components in the chain following the time order. If the fault manifestation time of the next component is close to the pinpointed component (e.g., time difference ≤ 2 seconds), we infer that the component's abnormal behavior is probably not caused by the anomaly propagation but a concurrent fault that occurs at a similar time. Thus, we will pinpoint this component as one faulty component as well.

If all the application components contain fault manifestations and the changes at all the components follow the same upward or downward trend, FChain infers that the performance anomaly is probably caused by some external factors such as workload increases (i.e., upward trend) or a NFS server problem (i.e., downward trend). In this case, FChain will not pinpoint any component within the application as faulty. Although previous work [22] has also addressed the problem of distinguishing workload changes from anomalies within a single component, our work provides workload change detection for distributed applications.

We may derive spurious abnormal change propagations between independent components. For example, in Figure 5, we will derive an abnormal change propagation path from the application server 1 to the application server 2 based on the timestamps of their abnormal change points. However, this propagation actually does not exist. We propose to use inter-component dependency relationships to filter out spurious propagation paths. For each suspicious component that contains the abnormal change point, we examine whether there is a path in the dependency graph from any pinpointed faulty components to this component. If no path in the dependency graph can be found, we pinpoint this component as a faulty one since the anomaly propagation is unlikely and the component's anomalous behavior must have been caused by an independent fault. We leverage previous black-box dependency discovery tools [11] to discover inter-component dependencies.[3]

However, we cannot solely rely on the dependency information for fault localization since the abnormal change propagation does not always follow the dependency path. For example, in RUBiS, the faulty application server can cause its upstream component (the web server) to exhibit abnormal behavior due to a *back-pressure* effect, that is, a faulty component might cause its upstream component to show anomalous behavior.[4] If we only rely on the dependency information, we will pinpoint the normal web server as the faulty component and miss the true culprit component, that is, the application server. Our experimental results in Section III will confirm this observation.

We also found that existing network trace based depen-

[3]To achieve high accuracy, the black-box dependency scheme needs to accumulate sufficient amount of network trace data [11]. Luckily, the application dependency information rarely change during application runtime. We perform the dependency discovery offline and store the results in a file for later reference.

[4]The cause of the back-pressure varies among different applications. One common reason is that after the input buffer of the faulty component becomes full, it forces the upstream component to drop data or pause processing.

dency discovery scheme fails to discover any dependency information in the data stream processing system [14]. The reason is that the dependency discovery algorithm relies on the gap between network packets to separate network flows. However, the stream application processes continuous data packets, which do not contain gaps between network packets. Note that FChain can still pinpoint faulty components based on the abnormal change propagation paths when the dependency information is unavailable. In contrast, the dependency-only scheme will fail the fault localization task for distributed stream processing systems.

III. Experimental Evaluation

We have implemented the FChain system on top of the Xen platform [23], and conducted extensive experiments using the RUBiS multi-tier online auction benchmark (EJB version) [16], the IBM System S data stream processing system [14], and the Hadoop MapReduce framework [15]. In this section, we first describe our evaluation methodology followed by the experiment results.

A. Evaluation Methodology

Our experiments were conducted on the NCSU's Virtual Computing Lab (VCL), a production cloud computing infrastructure that operates in a similar ways as Amazon EC2 [1]. All the VCL hosts used in our experiments have a dual-core Xeon 3.00GHz CPU, 4GB memory, and 30GB disk, which are connected to Gigabit networks. Each host runs 64 bit CentOS 5.2 with Xen 3.0.3. The guest VMs also run 64 bit CentOS 5.2. FChain monitors each guest VM from Domain 0 using the `libxenstat` and `libvirt` libraries. Monitored metrics are cpu usage, memory usage, network in, network out, disk read, and disk write. The metric sampling interval is 1 second. To evaluate FChain in multi-tenant cloud computing environments, we run three benchmark systems concurrently on the same set of VCL hosts. We first describe the benchmark systems used in our experiments as follows.

RUBiS online auction benchmark: We use the three-tier online auction benchmark system RUBiS (EJB version). The topology of the RUBiS system is shown in Figure 5. We run each application component in one guest VM. In order to evaluate our system under workloads with realistic time variations, we use a client workload generator that emulates the workload intensity observed in the NASA web server trace beginning at 00:00:00 July 1, 1995 from the IRCache Internet traffic archive [24] to modulate the request rate of our RUBiS benchmark. The client workload generator also tracks the response time of the HTTP requests it made. An SLO violation is marked if the average request response time is larger than 100ms.

Hadoop: We run Hadoop sorting application, one of the sample applications provided by the Hadoop distribution. The application consists of three map nodes and six reduce nodes. The data size we process is 12GB, which is generated using the RandomWriter application. We measure the progress score of the job by calling the Hadoop API. An SLO violation is marked when the job does not make any progress for more than 30 seconds.

IBM System S: We use the commercial high-performance data stream processing system [14], System S, developed by IBM. In our experiments, we used a tax calculation application, one of the sample applications provided by System S product distribution. The topology of the System S application is shown in Figure 2. Each PE runs in a separate guest VM. In order to evaluate our system under workloads with realistic time variations, we used the workload intensity observed in the ClarkNet web server trace beginning at 1995-08-28:00.00 from the IRCache Internet traffic archive [24] to modulate the data arrival rate. We measured the average per-tuple processing time and an SLO violation is marked if the average processing time is larger than a pre-defined threshold (e.g., 20ms).

Fault injection. We inject different faults (e.g., common software bugs, bottleneck) during an application runtime. Each application run lasts one hour. We inject one fault at a random time instant to test FChain under different workload conditions. For each fault, we use 30 to 40 application runs. We test both single-component faults and multi-component concurrent faults.

For RUBiS, single-component faults include: 1) *MemLeak*: we start the program that has a memory leak bug in the VM running the database server; 2) *CpuHog*: a CPU-bound program competed CPU with the database server inside the same VM; and 3) *NetHog*: we use httperf [25] tool to send a large number of HTTP requests to the web server. Multi-component concurrent faults include: 1) *OffloadBug*: the application server 1 wants to offload some EJBs to the application server 2. However, the program bug (JIRA #JBAS-1442) in the application server 1 makes the remote server lookup return the local server binding by mistake; and 2) *LBBug*: a load balancing bug (mod_jk 1.2.30) causes the web server to dispatch requests unevenly. These two faults are real software bugs found in the JBoss and Apache load balancer software.

For Hadoop, we injected concurrent faults in all the map nodes: 1) *Concurrent MemLeak*: we injected a memory leak bug into all the map tasks, which allocated memory from the heap without releasing; 2) *Concurrent CpuHog*: we injected an infinite loop bug in all the map tasks; and 3) *Concurrent DiskHog*: we start a disk I/O intensive program in the Domain 0 of each host running the map tasks.

For System S, we inject the following single-component faults: 1)*MemLeak*: We inject a small snippet of code that has a memory leak bug into a randomly selected PE; 2) *CpuHog*: a CPU-bound program competes CPU with a PE within the same VM; 3) *Bottleneck*: we make one randomly selected PE the bottleneck by setting a low CPU cap over the PE. The multi-component concurrent faults include: 1) *Concurrent MemLeak*: we start the memory leak program simultaneously in two randomly selected PEs; and 2) *Concurrent CpuHog*: we start the CPU intensive program simultaneously in two randomly selected PEs.

We compare FChain with a set of existing black-box fault

localization schemes:

1) Histogram: This scheme computes an anomaly score for each system-level metric using Kullback-Leibler divergence [26] between the histogram of the most recent data contained in the same look-back window as FChain and the histogram of the whole data. It then pinpoints abnormal components based on the anomaly scores. We vary the anomaly score threshold to show the tradeoff between the true positive rate and the false positive rate. This scheme has been used by previous work for detecting anomalies (e.g., [10]).

2) NetMedic [9]: It is a recently developed application-agnostic multi-metric fault localization tool. The abnormal component pinpointing is based on the application topology and the inter-component impact learned from the historical data. This scheme needs to assume the knowledge of the application topology. For estimating the inter-component impact, we use the same 1800 seconds of recent data as specified in [9]. Different from FChain, NetMedic just gives a ranked list of all components based on their likelihood of being the faulty components. We first pinpoint the top impact component as the faulty component. We also pinpoint the following components whose impact difference with the top ranked component is less than a certain threshold δ. We adjust the value of δ to show the tradeoff between the true positive rate and the false positive rate that can be achieved by NetMedic.

3) Topology: This scheme assumes the knowledge of the application topology. It first detects abnormal components using the outlier change point detection algorithm developed in our previous work PAL [13]. It then pinpoints faulty components based on the topology information, that is, if the abnormal component C_2 depends on the abnormal component C_1, we pinpoint C_1 as the faulty component. By comparing FChain with this scheme, we want to show that it is insufficient to just consider the application topology for pinpointing faulty components.

4) Dependency: Instead of assuming the application topology knowledge, this scheme uses the black-box dependency discovery tool [11] to dynamically extract the inter-component dependency information. It first detects abnormal components using the same outlier change point detection algorithm as the Topology scheme. It then pinpoints faulty components based on the discovered dependency information. If no dependency information is discovered, this scheme will output all the components that have outlier change points as faulty components. By comparing FChain with this scheme, we want to show that it is insufficient to just rely on the dependency information for pinpointing faulty components.

5) PAL [13]: This is the initial version of our change propagation based fault localization system. However, different from FChain, PAL does not perform predictability-based abnormal change point selection or consider the dependency information in fault localization. It also does not support online validation.

6) Fixed-Filtering: This scheme uses the same pinpointing algorithm as FChain except that it employs a fixed prediction error filtering threshold to select the abnormal change points. We varied the filtering threshold to show different accuracy results that can be achieved by this scheme. We compare FChain with this fixed filtering scheme to show the effectiveness of our burst based filtering scheme.

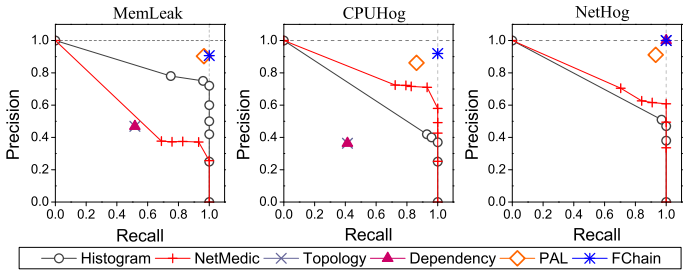

Fig. 6. Fault localization accuracy comparison for the single-component faults for RUBiS.

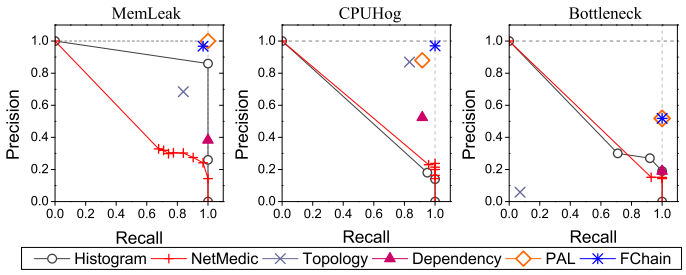

Fig. 7. Fault localization accuracy comparison for the single-component faults for System S.

To quantify the accuracy of different fault localization schemes, we use the standard precision and recall metrics. Let N_{tp}, N_{fn} and N_{fp} denote the number of true positives (correctly pinpoint a faulty component), false negatives (miss a faulty component), and false positives (pinpoint a normal component as faulty), respectively. We calculate the precision and recall metrics in the standard way as follows,

$$Precision = \frac{N_{tp}}{N_{tp} + N_{fp}}, Recall = \frac{N_{tp}}{N_{tp} + N_{fn}} \quad (1)$$

We evaluate the accuracy of different pinpointing algorithms using the commonly used "receiver operating characteristic" (ROC) curve whose X-axis and Y-axis show the recall and precision, respectively. A perfect pinpointing scheme should achieve 100% precision and 100% recall.

In our experiments, we configure the FChain system as follows. We set the look-back window (W) to be 100 seconds for all the tested faults except the DiskHog fault in the Hadoop application. The reason is that the DiskHog fault takes much longer time to manifest than the other faults. The 100 seconds look-back window will not cover the initial stage of the fault manifestation. Thus, we used a longer look-back window (500 seconds) for the DiskHog fault. We use a concurrency threshold of 2 seconds to classify concurrent faults, that is, if the abnormal change point time difference between two components is less than 2 seconds, we consider these two components as concurrent faulty components. The burst extraction window Q is set as 20 seconds. We use the

top 90% frequencies to synthesize the burst signal and use the 90th percentile of the burst value as the expected prediction error. We found those parameter configurations work well for all the applications tested in our experiments. We also conduct sensitivity study on those parameters and will show the results in Section III-F.

B. Single-Component Fault Localization Results

Figure 6 shows the pinpointing accuracy results for RUBiS under three single-component faults. We observe that FChain consistently achieves the highest precision and recall for all the faults. We observe that the Histogram scheme does not work well for the CpuHog and NetHog faults that manifest very quickly. The reason is that when the performance anomaly is detected, the histogram of the recent data has not shown significant difference from the histogram of all historical data yet since the fault manifestation duration is very short. The histogram scheme works better for gradually changing faults such as memory leak although it is still less accurate than FChain. We observe that NetMedic could not achieve high accuracy during this set of experiments. After examining the logs, we found that the pinpointing errors are caused by unseen states that make the system assign inaccurate impact values.[5] During the fault injection, the faulty component and the other affected components often have a state that is not present in the historical data. In contrast, we observe that FChain is not susceptible to the problem of unseen values.

By employing predictability-based filtering and leveraging dependency information, FChain effectively removes irrelevant change points caused by normal workload fluctuations. Thus, FChain can achieve higher accuracy than the other change point based schemes such as Topology, Dependency, and PAL. Since the dependency discovery scheme accurately identifies all the dependencies in the RUBiS system, the Dependency scheme has the same accuracy as the Topology scheme in this case. Particularly, the Topology and the Dependency schemes have very low accuracy for the MemHog and CpuHog faults. The reason is that we injected those two faults at the database server that is the last tier in the RUBiS system. We observed the "back-pressure" symptom mentioned in Section II-C. The faulty database server causes its upstream component (the web server or the application server) to exhibit anomalous behaviors. If we perform pinpointing based on the dependency or topology, we will mistakenly pinpoint the upstream component of the culprit component as the faulty one. We injected the NetHog fault in the web server that is the first tier in RUBiS. Thus, both Topology and Dependency perform well since the back-pressure problem does not exist. In contrast, FChain is not sensitive to the location of the faulty component, which can achieve high accuracy for all situations. Although FChain also considers the dependency information, we observe that when a fault propagates back to the upstream components, its impact becomes smaller. The abnormal change point selection step can effectively filter out those change points.

[5]NetMedic assigns a default high impact value (0.8) to an edge connecting to the abnormal component with a previously unseen state.

Figure 7 shows the fault localization accuracy comparison results for the System S single component faults. Similar to the RUBiS experiments, FChain consistently achieves the highest precision and recall values for all the tested faults. The dependency discovery scheme fails to detect any dependency relationship for System S due to the reason mentioned in Section II-C. Thus, the Dependency scheme pinpointed all the components that have outlier change points. This is reason why the Dependency scheme has low precision for all the cases. The Topology scheme does not perform well for the MemHog and the bottleneck faults because of the same back-pressure problem mentioned in the RUBiS results. We also observe that all the schemes have low precision for the bottleneck fault. The reason is that the fault propagates very quickly due to high-throughput communication between stream processing components. Thus, it is difficult to distinguish single-component faults from concurrent multi-component faults, which explains why the precision is low. Luckily, we can use the online validation to quickly remove those false alarms, which will be shown in Section III-D.

C. Multi-Component Concurrent Fault Localization Results

We now evaluate FChain using multi-component concurrent faults. Figures 8, 9, and 10 show the pinpointing accuracy results for RUBiS, System S, and Hadoop, respectively.

We observe that FChain consistently achieves high precision and recall results in all the tested cases except the concurrent CPUHog in System S. After examining the log file, we find the diagnosis errors are mostly caused by the side-effect of smoothing. Although our previous work [13] showed that smoothing helps to remove the random noise in the raw monitoring data, smoothing in this case causes the time of the abnormal change point in the affected normal component to become earlier than those of true culprit components. We need to perform adaptive smoothing to address this problem, which is part of our on-going work.

Compared to RUBiS and System S, Hadoop is much more dynamic with highly fluctuating system metrics. In this case, the simple change point detection schemes such as PAL have low accuracy, especially for the CpuHog and DiskHog faults. In the Hadoop experiments, we inject faults into all the map nodes that are the first components in the topology order. The "back-pressure" problem does not exist in this case. This explains why Topology and Dependency achieve high accuracy. NetMedic also achieves high precision and recall values in the MemLeak and CPUHog faults. The reason is that the default high impact value for unseen states happen to be correct. However, for the DiskHog fault, the default high impact value is incorrect, which causes NetMedic to have low accuracy. In contrast, FChain can handle previously unseen values and consistently achieve high accuracy.

D. Online Pinpointing Validation Results

We now evaluate our online validation scheme. We pick two most challenging faults where all the schemes do not perform well. They are the Bottleneck fault and the concurrent CpuHog

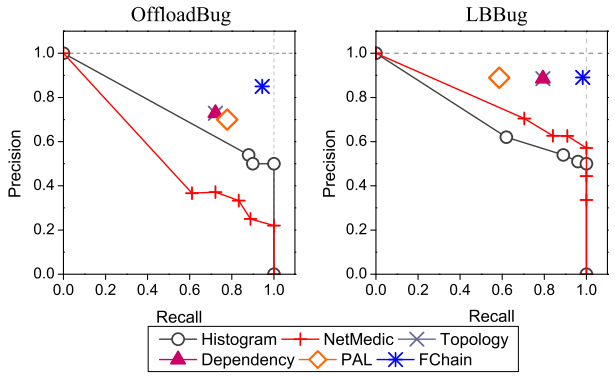

Fig. 8. Fault localization accuracy comparison for the multi-component faults in RUBiS.

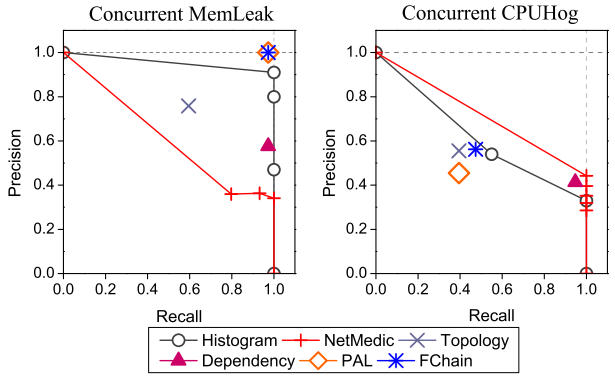

Fig. 9. Fault localization accuracy comparison for the multi-component faults in System S.

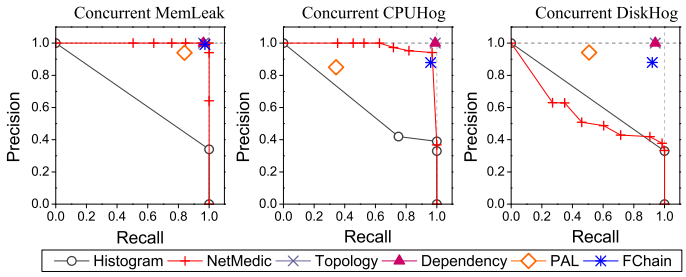

Fig. 10. Fault localization accuracy comparison for the multi-component faults in Hadoop.

fault in System S. Figure 11 shows the pinpointing accuracy results for different schemes. The "FChain+VAL" denotes the FChain scheme with the online pinpointing validation activated. Note that the results for the FChain scheme shown before are the results achieved by FChain without the online validation. We observe that our online validation scheme can successfully remove most false alarms for these two faults. FChain can quickly identify the true faulty component(s) by properly scaling the right resource metric. However, our current online validation scheme cannot help to improve the recall value, which is part of our on-going work.

E. Comparison with Fixed Filtering Schemes

We now compare FChain with the Fixed-Filtering scheme. Due to space limitations, we only show a subset of our results.

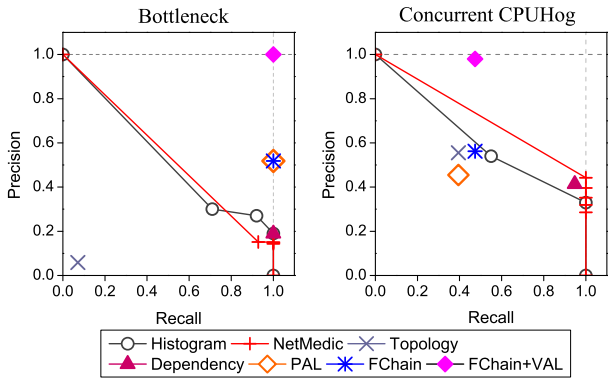

Fig. 11. Online validation effectiveness for two challenging System S faults.

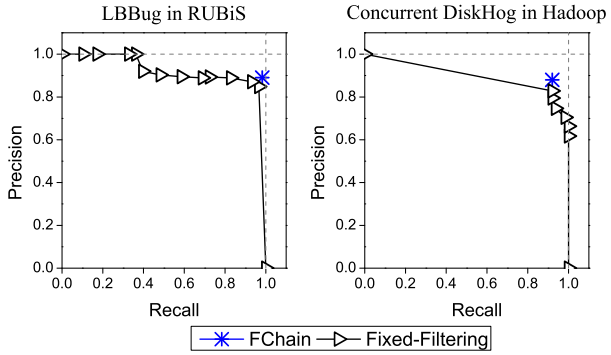

Fig. 12. Fault localization accuracy comparison with the Fixed-Filtering scheme for LBBug in RUBiS and DiskHog in Hadoop.

Figure 12 shows the accuracy of the Fixed-Filtering scheme for a subset of faults in RUBiS and Hadoop. We observe that the Fixed-Filtering scheme is very sensitive to the prediction error filtering threshold value. In contrast, FChain can automatically infer the optimal (or near optimal) filtering threshold based on the burstiness in the metric values.

F. Sensitivity Study

FChain parameters		**NetHog RUBiS**	**CPUHog System S**	**Diskhog Hadoop**
Look-back window W (sec)	**100**	**P=1, R=1**	**P=0.97, R=1**	P=0.56, R=0.63
	300	P=0.98, R=1	P=0.95, R=0.95	P=0.88, R=0.9
	500	P=0.98, R=1	P=0.92, R=0.95	**P=0.88, R=0.92**
Concurrency threhold (sec)	**2**	**P=1, R=1**	**P=0.97, R=1**	**P=0.88, R=0.92**
	5	P=1, R=1	P=0.93, R=1	P=0.88, R=0.88
	10	P=0.97, R=1	P=0.93, R=1	P=0.83, R=0.88

TABLE I
PRECISION (P) AND RECALL (R) VALUES UNDER DIFFERENT CONFIGURATIONS OF THE KEY FCHAIN PARAMETERS.

We have conducted sensitivity study to evaluate the impact of key FChain system parameters to its pinpointing accuracy. Due to space limitations, we only show a subset of our results in Table I with the optimal parameter settings highlighted in bold. Overall, we found that FChain is not sensitive to different parameter values. We observe that FChain can achieve the optimal performance using default setting (100 seconds look-back window, 2 seconds concurrency threshold) for all the tested faults except one case that is the look-back window

System Modules	CPU cost
VM monitoring (6 attributes)	1.03±0.09 milliseconds
Normal fluctuation modeling (1000 samples)	22.9 ± 2 milliseconds
Abnormal change point selection (100 samples)	602.4 ± 105.2 milliseconds
Integrated fault diagnosis	22 ± 1 microseconds
Online validation (per-component)	30 ± 1 seconds

TABLE II
FCHAIN OVERHEAD MEASUREMENTS.

size for the DiskHog fault in Hadoop. The reason has been described in Section III-A. Generally, the look-back window should be long enough to capture the fault manifestation. We are currently investigating an adaptive look-back window configuration scheme by examining the metric changing speed.

G. FChain System Overhead Measurements

We now evaluate the overhead of the FChain system. Table II lists the CPU cost of each key module in FChain. We observe that most modules in FChain is light-weight. The most computation-intensive module is the abnormal change point selection component, which is triggered only when a performance anomaly occurs. FChain also distributes the change point computation load on different hosts and executes them in parallel to achieve scalability. The online validation takes about 30 seconds for each component since we need some time to observe scaling impact for deciding whether we have made a pinpointing error. However, the online validation is only performed on those suspicious components pinpointed by the integrated fault diagnosis module. The FChain daemon running inside the Domain 0 of each host imposes less than 1% CPU load and consumes about 3MB memory during normal execution.

IV. RELATED WORK

Our work is first closely related to previous black-box fault localization schemes. For example, NetMedic [9] provided detailed application-agnostic fault diagnosis by learning inter-component impact. NetMedic first needs to assume the knowledge of the application topology. To perform impact estimation, NetMedic needs to find a historical state that is similar to the current state for each component. However, for previously unseen anomalies, we might not be able to find a historical state that is similar to the current state for the faulty components. Under those circumstances, NetMedic assign a default high impact value, which sometimes lead to inaccurate diagnosis results as shown in Section III. In comparison, FChain can diagnose previously unseen anomalies and does not assume any prior application knowledge. Oliner et al. [10] proposed to compute anomaly scores using the histogram approach and correlates the anomaly scores of different components to infer the inter-component influence graph. As shown in Section III, it is difficult for the histogram-based anomaly detection to perform online fault localization over suddenly manifesting faults. Moreover, unrelated components can have indirect correlations caused by workload fluctuations, which will cause their system to raise false alarms. In comparison, FChain is more robust to different types of faults and workload fluctuations.

To achieve black-box diagnosis, researchers have also explored various passive network traffic monitoring and analysis techniques such as Sherlock [11], Orion [27], SNAP [28]. However, those analysis schemes can only achieve coarse-grained machine-level fault localization. Additionally, during our experiments, we found that previous network trace analysis techniques cannot handle continuous data stream processing applications due to the lack of gaps between packets for extracting different network flows. Project5 [29] and E2EProf [30] performed cross-correlations between message traces to derive causal paths in multi-tier distributed systems. WAP5 [31] extends the black-box causal path analysis to support wide-area distributed systems. Orion [27] discovers dependencies from network traffic using packet headers and timing information based on the observation that the traffic delay distribution between dependent services often exhibits typical spikes. LWT [32] proposed to discover the similarity of the CPU usage patterns between different VMs to extract the dependency relationships between different VMs. However, as shown in our experiments, dependency-based fault localization techniques are not robust, which can make frequent pinpointing mistakes due to various reasons (e.g., the "back pressure" effect in distributed applications, common network services pinpointed as culprits). Furthermore, existing dependency discovery techniques need to accumulate a large amount of trace data to achieve reasonable accuracy. Particularly, network trace based techniques only support request-and-reply types of applications, which fail to discover any dependency in continuously running applications such as data stream processing systems. In contrast, FChain provides online fault localization, which does not require any training data for anomalies or a large amount of training data for normal behaviors. FChain is fast, which can quickly localize faulty components with high accuracy after the performance anomaly is detected.

A flurry of research work has proposed to use end-to-end tracing for distributed system debugging. Magpie [3] is a request extraction and workload modelling tool that can record fine-grained system events and correlate those events using an application specific event schema to capture the control flow and resource consumption of each request. Pinpoint [4] takes a request-oriented approach to tag each call with a request ID by modifying middleware platform and applies statistical methods to identify components that are highly correlated with failed requests. Monitor [33] tracks the requests exchanged between components in the system and performs probabilistic diagnosis on the potential anomalous components. X-Trace [5] is an integrated cross-layer, cross-application tracing framework, which tags all network operations resulting from a particular task with the same task identifier to construct a task tree. Spectroscope [34] can diagnose performance anomalies by comparing request flows from two executions. In contrast, our approach does not require any instrumentation to the

application or middleware platform to collect request flows. Thus, it is much easier to deploy FChain in large-scale IaaS clouds.

Blacksheep [35] correlates the change point of system-level metrics (e.g., cpu usage) with the change in count of Hadoop application states (i.e., events extracted from logs of DataNodes and TaskTrackers) to detect and diagnose the anomalies in a Hadoop cluster. Kahuna-BB [36] correlates black-box data (system-level metrics) and white-box data (Hadoop console logs) across different nodes of a MapReduce cluster to identify faulty nodes. In comparison, FChain is a black-box fault localization system, which is application-agnostic without requiring any knowledge about the application internals. We believe that FChain is more practical and attractive for IaaS cloud systems than previous white-box or gray-box techniques.

V. Conclusion

In this paper, we have presented FChain, a robust black-box online fault localization system for IaaS cloud computing infrastructures. FChain can quickly pinpoint faulty components immediately after the performance anomaly is detected. FChain provides a novel predictability-based abnormal change point selection scheme that can accurately identify the onset time of the abnormal behaviors at different components processing dynamic workloads. FChain combines both the abnormal change propagation knowledge and the inter-component dependency information to achieve robust fault localization. FChain can further remove false alarms by performing online validation. We have implemented FChain on top of the Xen platform and conducted extensive experimental evaluation using IBM System S data stream processing system, Hadoop, and RUBiS online auction benchmark. Our experimental results show that FChain can achieve much higher accuracy (i.e., up to 90% higher precision and up to 20% higher recall) than existing schemes. FChain is light-weight and non-intrusive, which makes it practical for large-scale IaaS cloud computing infrastructures.

Acknowledgment

This work was sponsored in part by NSF CNS0915567 grant, NSF CNS0915861 grant, NSF CAREER Award CNS1149445, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, IBM Faculty Awards and Google Research Awards. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, ARO, or U.S. Government. The authors would like to thank the anonymous reviewers for their insightful comments.

References

[1] "Amazon Elastic Compute Cloud," http://aws.amazon.com/ec2/.
[2] "Virtual computing lab," http://vcl.ncsu.edu/.
[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *OSDI*, 2004.
[4] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," in *NSDI*, 2004.
[5] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A pervasive network tracing framework," in *NSDI*, 2007.
[6] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control," in *OSDI*, 2004.
[7] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *SOSP*, 2005.
[8] S. Duan, S. Babu, and K. Munagala, "Fa: A system for automating failure diagnosis," in *ICDE*, 2009.
[9] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and V. Bahl, "Detailed diagnosis in computer networks," in *SIGCOMM*, 2009.
[10] A. J. Oliner, A. V. Kulkarni, and A. Aiken, "Using correlated surprise to infer shared influence," in *DSN*, 2010.
[11] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *SIGCOMM*, 2007.
[12] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive Elastic ReSource Scaling for Cloud Systems," in *CNSM*, 2010.
[13] H. Nguyen, Y. Tan, and X. Gu, "PAL: Propagation-aware anomaly localization for cloud hosted distributed applications," in *SLAML*, 2011.
[14] B. Gedik, H. Andrade, K. L. Wu, P. S. Yu, and M. Doo, "SPADE: the system s declarative stream processing engine," in *SIGMOD*, 2008.
[15] "Apache Hadoop System," http://hadoop.apache.org/core/.
[16] "Rice university bidding system," http://rubis.objectweb.org/.
[17] M. Ben-Yehuda, D. Breitgand, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg, "NAP: a building block for remediating performance bottlenecks via black box network analysis," in *ICAC*, 2009.
[18] Y. Tan, X. Gu, and H. Wang, "Adaptive system anomaly prediction for large-scale hosting infrastructures," in *PODC*, 2010.
[19] D. L. Mills, "A brief history of NTP time: memoirs of an internet timekeeper," *Computer Communication Review*, 2003.
[20] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems," in *ICDCS*, 2012.
[21] M. Basseville and I. V. Nikiforov, *Detection of abrupt changes: theory and application*. Prentice-Hall, Inc., 1993.
[22] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *DSN*, 2008.
[23] P. Barham and et al., "Xen and the art of virtualization," in *SOSP*, 2003.
[24] "The ircache project," http://www.ircache.net/.
[25] "Httperf," http://code.google.com/p/httperf/.
[26] S. Kullback, "The kullback-leibler distance," *The American Statistician*, 1987.
[27] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating network application dependency discovery: experiences, limitations, and new solutions," in *OSDI*, 2008.
[28] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim, "Profiling network performance for multi-tier data center applications," in *NSDI*, 2011.
[29] M. K. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *SOSP*, 2003.
[30] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham, "E2EProf: Automated end-to-end performance management for enterprise systems," in *DSN*, 2007.
[31] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat., "WAP5: black-box performance debugging for wide-area systems," in *WWW*, 2006.
[32] R. Apte, L. Hu, K. Schwan, and A. Ghosh, "Look Who's Talking: Discovering dependencies between virtual machines using cpu utilization," in *HotCloud*, 2010.
[33] G. Khanna, I. Laguna, F. Arshad, and S. Bagchi, "Distributed diagnosis of failures in a three tier e-commerce system," in *SRDS*, 2007.
[34] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *NSDI*, 2011.
[35] J. Tan and P. Narasimhan, "RAMS and BlackSheep: Inferring white-box application behavior using black-box techniques," CMU, Tech. Rep., 2008.
[36] J. Tan, X. Pan, E. Marinelli, S. Kavulya, R. Gandhi, and P. Narasimhan, "Kahuna: Problem diagnosis for mapreduce-based cloud computing environments," in *NOMS*, 2010.